Assembler IA16

8086

- **Anno 1978**
- **Bus Indirizzi a 20 bit**
- **Bus Dati a 16 bit**

8088

- **Anno 1979**
- **Bus Indirizzi a 20 bit**
- **Bus Dati a 8 bit**
- **Set di istruzioni compatibile con 8086**

Registri 8086

Registri Dati

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Puntatori

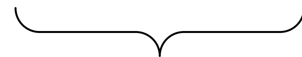
SP
BP
SI
DI

Registri di Segmento

CS
SS
DS
ES

IP

FLAGS



16 bit

Registri di Dati

Possono essere impiegati come registri a **8 bit** o **16 bit.**

Come registri a **16 bit** vengono indicati come **AX,BX,CX,DX.**

Come registri a **8 bit** vengono indicati come **AH,AL,BH,BL,CH,CL,DH,DL.**

Possono essere utilizzati in operazioni aritmetiche e logiche. Ad alcuni di questi registri è riservato un ruolo specifico in determinate istruzioni.

AX (Accumulatore)

Utilizzato nelle istruzioni aritmetiche, di I/O e con stringhe

BX (Base Register)

Può essere utilizzato come registro base per il calcolo di indirizzi

CX (Count Register)
Utilizzato tipicamente come contatore

DX (Data Register)
Utilizzato in operazioni di I/O, moltiplicazione, divisione con numeri a 32 bit (in coppia con AX)

Registri Indice e Puntatori

Tali registri contengono generalmente lo scostamento all'interno di un segmento.

SP (Stack Pointer)
Puntatore alla cima dello stack

BP (Base Pointer)
Utilizzato come puntatore all'interno dello stack ma può essere impiegato amche come generico registro indice

SI (Source Index)
Registro sorgente o come g*enerico registro indice.*

DI (Destination Index)
Registro destinazione o come g*enerico registro indice.*

Registro Flag

-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Flag di Stato

AF (Auxiliary Flag): indica una condizione di riporto durante un'operazione BCD

CF (Carry Flag): indica una condizione di riporto durante un'operazione aritmetica

OF (Overflow Flag): indica una condizione di overflow durante un'operazione aritmetica

SF (Sign Flag): indica il segno del risultato di un'operazione

PF (Parity Flag): indica il bit di parita' del risultato

ZF (Zero Flag): indica se il il risultato di un'operazione e' zero

Flag di Controllo

IF (Interrupt Enable Flag): abilita/disabilita gli interrupt mascherabili

TF (Trap Flag): pone il processore in modalita' single step (debugging).
Ad ogni istruzione viene generato un interrupt via software

DF (Direction Flag): indica la direzione sulla quale operare sulle stringhe

Organizzazione della memoria

L'8086 ha una architettura con memoria segmentata e segmenti da 64KB.

Per l'indirizzamento di **1 MB** sono necessarie **20 linee di indirizzo** (**indirizzo fisico**). I **registri dell'8086** sono a **16 bit** e sono in grado di indirizzare solo 64k di memoria.

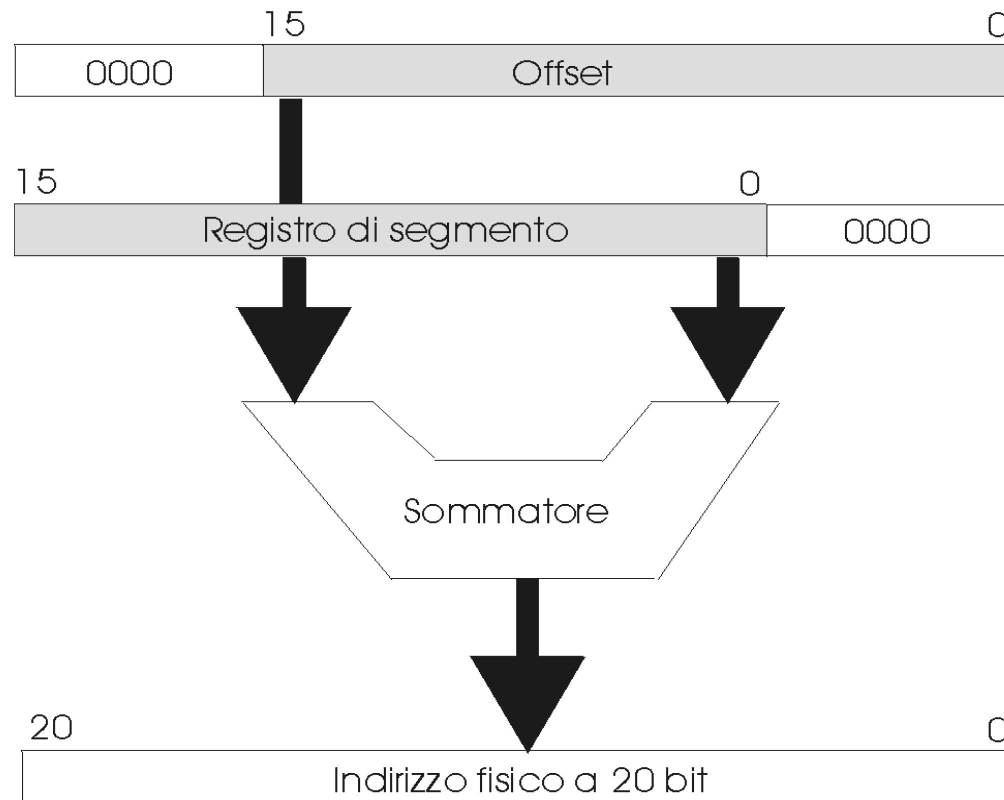
Per potere indirizzare l'intero Mega Byte si è introdotta la **rappresentazione "logica" della memoria** che **fa uso di due parametri** da **16 bit:** con la **BASE** (o **SEGMENTO**) si indirizza una delle 64K aree, mentre con l'**OFFSET** (o **indirizzo effettivo, EA**) si identifica la cella all'interno dell'area (SEGMENTO) selezionata che contiene a sua volta 64K locazioni di memoria. La CPU in ogni istante è in grado di accedere un insieme di 4 segmenti aventi dimensione di 64KB.

Indirizzo Logico = (base, offset) = SEGMENTO:OFFSET

Quando il programmatore indica un indirizzo logico **il processore lo converte in indirizzo fisico** per accedere alla memoria:

Indirizzo Fisico(20 bit) = base * 16 + offset

(la moltiplicazione per 16 si ottiene aggiungendo "0000" alla base)



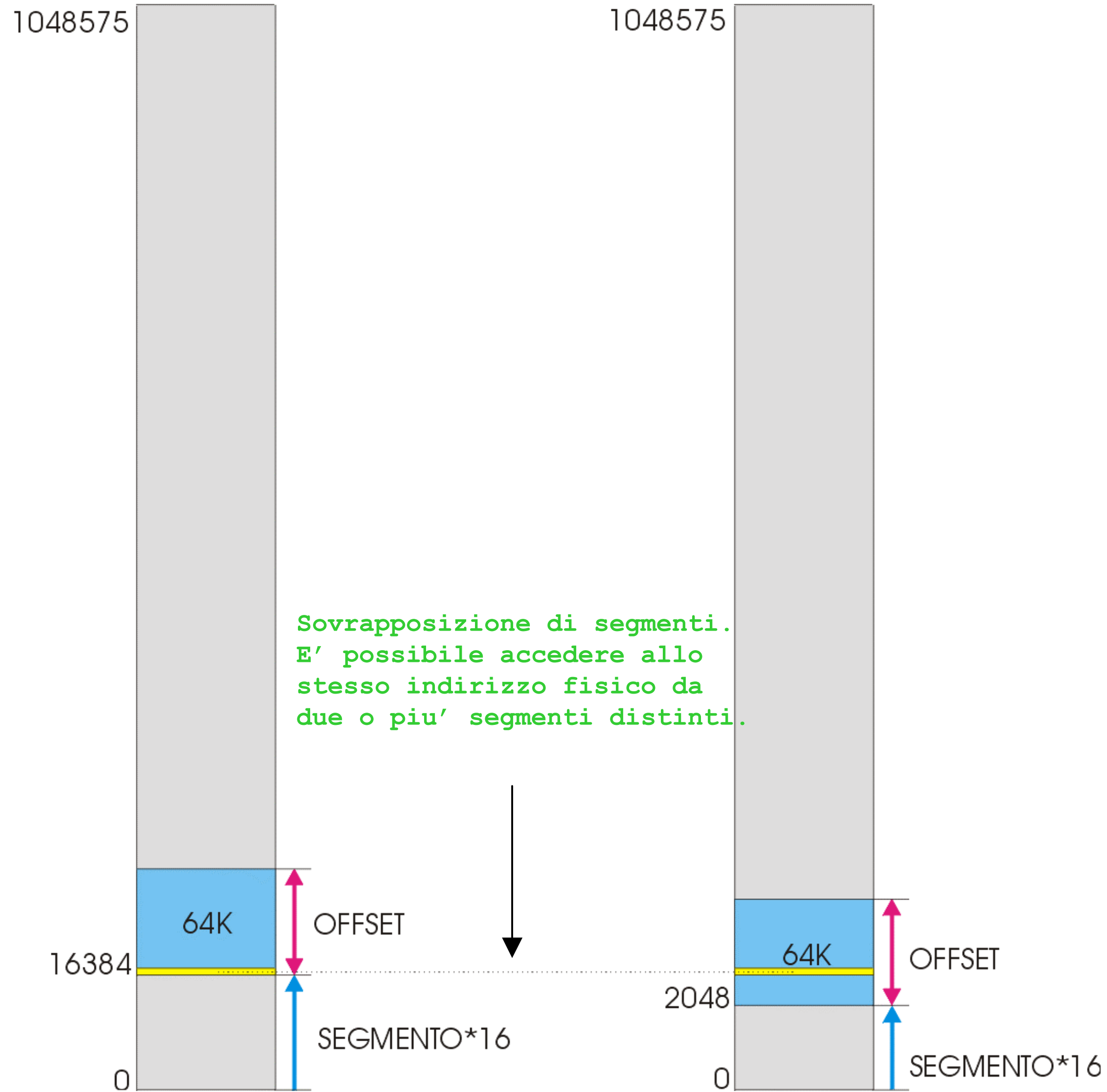
E' possibile accedere alla stessa cella di memoria da piu' segmenti differenti agendo sui parametri SEGMENTO e OFFSET (sovrapposizione di segmenti).

Esempio (in base 10) di sovrapposizione:

SEGMENTO:OFFSET 1024:0 -> 16384 = 1024*16 + 0

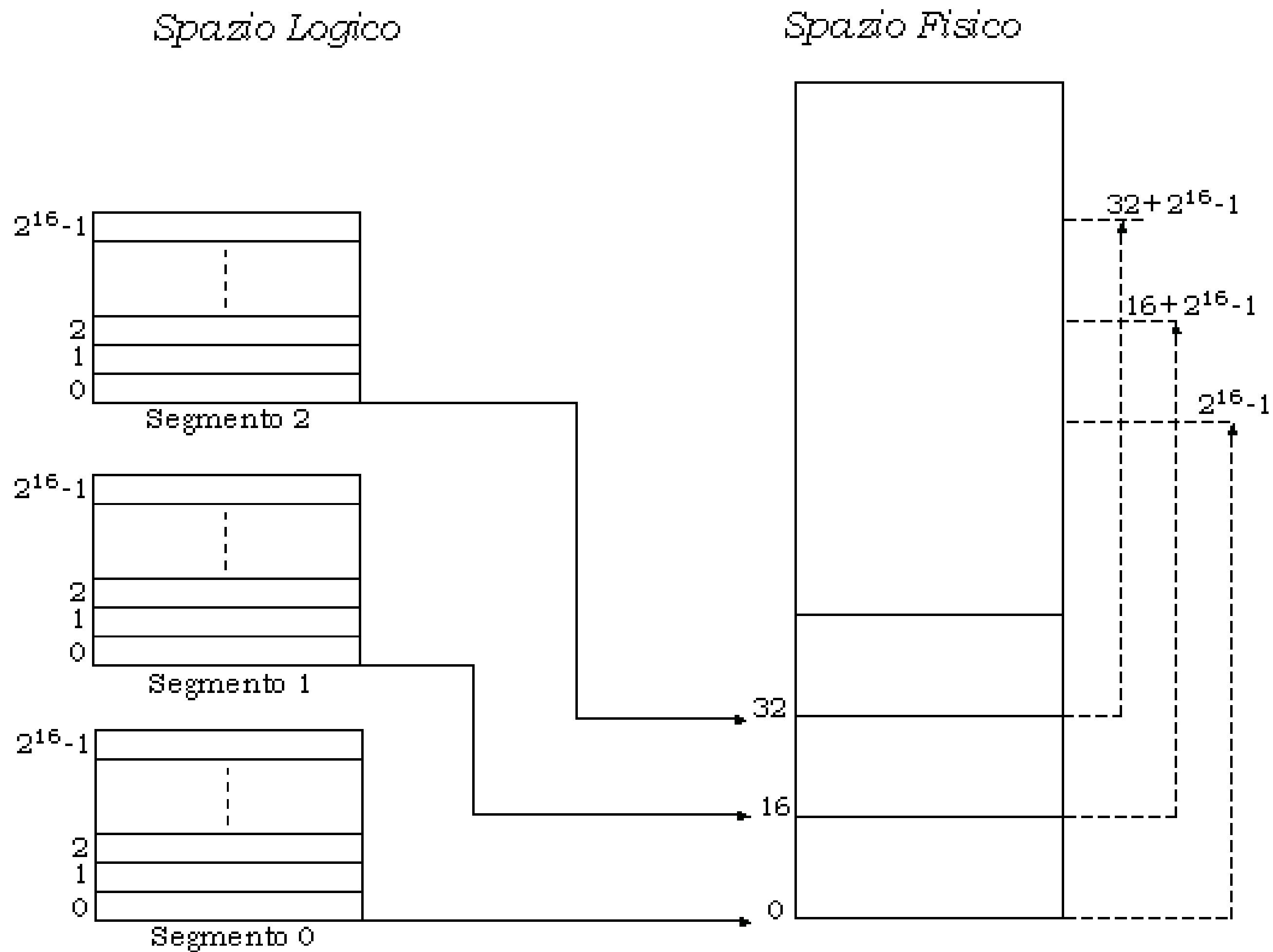
SEGMENTO:OFFSET 128:14336 -> 16384 = 128*16 + 14336

Indirizzo Logico **Indirizzo Fisico (20 bit)**



SEGMENTO:OFFSET 1024:0

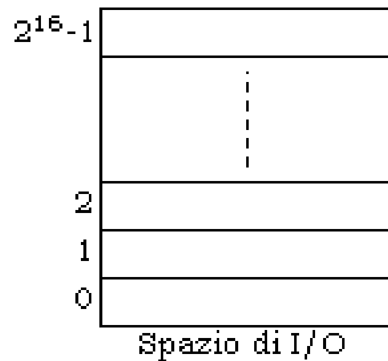
SEGMENTO:OFFSET 128:14336



All'interno dello spazio di indirizzamento d1 1 MB dell'8086 è possibile definire 65536 segmenti distinti.

In ogni istante possono essere utilizzati contemporaneamente solo **4** di questi segmenti: **Data Segment** (registro DS), **Code Segment** (registro CS), **Stack Segment** (registro SS) e **Extra Segment** (registro ES).

Organizzazione dello spazio di I/O



Lo **Spazio di I/O** ha una dimensione di 64K e non è segmentato:

Indirizzo Logico == Indirizzo Fisico

Sono disponibili 16 linee di indirizzo:

2^{16} (=64K) locazioni di I/O Spazio di I/O dette anche "**port**"

Registro IP (Instruction Pointer)

In ogni istante l'*instruction pointer* **IP** individua
la successiva istruzione da eseguire.

L'indirizzo di tale istruzione risulta:

CS:IP	**->**	**CS*16 + IP**
(**Indirizzo Logico**)		(**Indirizzo Fisico**)

I programmi non hanno accesso diretto al registro IP ma
il flusso delle istruzioni lo modifica implicitamente.

Nel caso di salti, procedure e *interrupt* i registri CS
e IP vengono modificati in modo da puntare all'indirizzo
di destinazione del salto o della procedura.

Linguaggio Macchina

L’insieme delle istruzioni che un elaboratore è in grado di eseguire. Tale linguaggio è strettamente correlato alla realizzazione fisica dell'elaboratore

Esempio di possibile istruzione macchina:

“***Copia*** ***il valore della variabile alfa nel registro BX***”

• Il campo **Op-code** specifica il tipo di operazione richiesta (es., “copia”)

• Il campo **Dest** specifica l’operando di destinazione (es., con 3 bit si individua uno degli 8 registri generali)

•Il campo **Source** specifica l’operando sorgente (es., l’indirizzo di alfa relativo al segmento dati)

Per scrivere codice in **linguaggio macchina** è necessario:

- Conoscere l'architettura della macchina
- Ragionare in termini del comportamento dell'elaboratore
- **Conoscere i dettagli relativi alla scrittura delle singole istruzioni:**
  - codici numerici e formato interno delle istruzioni macchina
  - rappresentazione degli operandi
  - gestire direttamente gli indirizzi in memoria per il riferimento ai dati e per i salti

Ad esempio: per scrivere nel linguaggio macchina dell'8086 l'istruzione:

MOV <destinazione>, <sorgente>

è necessario scegliere tra **14 diversi codici operativi, in funzione del tipo di operandi**

L'istruzione macchina che si ottiene ha una lunghezza variabile tra i **due e i sei byte**

Linguaggio Assembler

Come il linguaggio macchina e’ strettamente legato all’architettura della macchina (codice *non portabile*).

Consente un maggiore livello di astrazione rispetto al linguaggio macchina *nascondendo* i dettagli realizzativi delle singole istruzioni (codici operativi, formati, etc)

Caratteristiche del linguaggio Assembler

- Associa ai dati e alle istruzioni **nomi simbolici che identificano in modo univoco le corrispondenti posizioni di memoria** (eliminando, così, la necessità di utilizzare indirizzi espliciti).
- Consente utilizzare **istruzioni** in un linguaggio più intuitivo
- Fornisce **direttive** (o *pseudoistruzioni*) che controllano il comportamento dell'assemblatore in fase di traduzione facilitando lo sviluppo dei programmi permettendo:

  -la suddivisione di un'applicazione in più moduli

  - la definizione di macro, variabili, label, etc.

L'architettura della macchina a cui il linguaggio si riferisce non viene nascosta in alcun modo

Ad esempio, l'assembler consente di accedere direttamente ai registri, azione in genere preclusa dai linguaggi ad alto livello a causa della loro indipendenza dall’hardware.

Formato delle istruzioni

[label] istruzione/direttiva [operando/i] [; commento]

- **label:** consente di assegnare un **nome simbolico** a **variabili**, **valori**, **singole istruzioni**, **procedure**
- **istruzione/direttiva** è il **codice mnemonico per un'istruzione o una direttiva:**

  Nel caso di istruzioni individua il tipo di operazione da eseguire, il numero e il tipo degli operandi (la semantica dell'istruzione)
- **operando/i** è una combinazione di nessuna, una o più **costanti**, **registri riferimenti alla memoria;**

Per convenzione se un'istruzione ammette due operandi (**OP dst,src)** il primo è l'operando destinazione (dst),il secondo è l'operando sorgente (src).

Ad esempio,
MOV AX, CX copia il contenuto del registro CX nel registro AX

- **commento** consente di rendere più leggibile il programma

Il tipo di operandi ammessi varia da istruzione a istruzione.

Esistono istruzioni che ammettono come operando solo una costante o solo un particolare registro generale.

Esistono istruzioni con operandi impliciti.

Il set di istruzioni dell'8086 ***non ha la caratteristica della ortogonalità***

Nomi simbolici

- caratteri ammessi per i nomi simbolici:
  A-Z a-z 0-9 _ $?
- il primo carattere di un nome non può essere un digit (0-9)
- ogni nome **deve essere univoco** (in genere, all'interno del modulo in cui viene definito)
- **non deve coincidere con una parola riservata** (ad esempio, il nome di un registro o di un operatore non sono ammessi)

Costanti numeriche

- di *default*, tutti i valori numerici sono espressi **in base dieci**
- è possibile esprimere le costanti numeriche:
  - **in base 16** (esadecimale) mediante il suffisso **H** (il primo digit deve però essere numerico)
  - **in base 8** (octal) mediante il suffisso **O**
  - **in base 2** (binario) mediante il suffisso **B**

Ad esempio:

```
0FFH
0H
777O
11001B
FFH           errata !
778O          errata !
```

Direttive dell'Assembler

Istruzioni di tipo dichiarativo che possono essere interpretate dall'assemblatore.

Direttive per la definizione dei segmenti

Permettono di controllare in modo completo la definizione dei vari tipi di segmenti. La definizione di ogni segmento deve iniziare con una **direttiva SEGMENT** e deve terminare con una **direttiva ENDS:**

```
nomeSeg SEGMENT
contenuto del segmento
nomeSeg ENDS
```

La direttiva SEGMENT definisce l'inizio di un segmento.

La direttiva ENDS definisce la fine di un segmento.

Esempio - Per creare un programma con tre segmenti (stack, dati e codice), si scriverà:

```
_Stack SEGMENT  STACK  'STACK'
       dimensionamento dello stack
_Stack ENDS


_Data  SEGMENT  'DATA'
       definizione dei dati del programma
_Data  ENDS


_Code  SEGMENT  'CODE'
       ASSUME CS:_Code, SS:_Stack
       ASSUME ES:NOTHING
Start:
       MOV  AX, _Data
       MOV  DS, AX
       ASSUME DS:_Data
       . . .
_Code  ENDS
```

Direttive per la definizione di procedure

La definizione di ogni procedura deve iniziare con una direttiva PROC e deve terminare con una direttiva ENDP.

```
nome  PROC [distanza]
```

- **nome** è il **nome simbolico della procedura** (da utilizzare nelle chiamate alla procedura)
- **distanza** è:
  - ***NEAR*** - la procedura può essere chiamata solo all'interno del segmento in cui è stata definita (default)
  - ***FAR*** - la procedura può essere chiamata da qualsiasi segmento

La direttiva PROC definisce l'inizio di una procedura.

La direttiva ENDP definisce la fine di una procedura.

Ad esempio: per definire la procedura FAR di nome FarProc, si scrive:

```
FarProc PROC   FAR
        .......
        <instruzioni>
        .......
FarProc ENDP
```

Mentre per definire la procedura NEAR di nome NearProc, si scriverà:

```
NearProc        PROC   NEAR
        .......
        <instruzioni>
        .......
NearProc        ENDP
```

Direttive per la definizione di dati

Permettono di definire:

- **il nome**
- **il tipo**
- **il contenuto**

delle variabili in memoria.

```
[nome]  tipo  espressione [, espressione] ...
```

- **nome - nome simbolico del dato**

- **tipo - lunghezza del dato** (se scalare) o di ogni elemento del dato (se array) - i tipi più utilizzati sono:

DB	riserva uno o più byte (8 bit)
DW	riserva una o più word (16 bit)
DD	riserva una o più doubleword (32 bit)
…	

- **espressione - contenuto iniziale del dato:**

  - un' espressione costante
  - una stringa di caratteri (solo DB)
  - un punto di domanda (nessuna inizializzazione)
  - un'espressione che fornisce un indirizzo
  - un'espressione **DUP**licata

Esempi

```
ByteVar   DB 0          ;       1 byte inizializzato a 0

ByteArray DB 1,2,3,4    ,       ;array di 4 byte

Zeros      DB 256 dup(0)        ;array di 256 byte
                                ;inizializzati a 0
```

Direttiva per la definizione di costanti: EQU (EQUate)

In fase di '*assemblaggio*' del programma, questa direttiva permette di definire simboli che rappresentano valori specifici.

nome EQU espressione

Esempi:

```
K        EQU      1024
```

fa in modo che ogni volta che compare K nel codice questa sia considerata come fosse scritto "1024". Supponiamo di voler definire nomi di variabili dalla dimensione multipla di 1K (1024):

```
VETT1           DB      1024     DUP(?)
VETT2           DB      2*1024   DUP(?)
VETT3           DB      3*1024   DUP(?)
```

Se si definisce il simbolo K per mezzo di EUQ come visto sopra, potremo scrivere:

```
VETT1           DB      K        DUP(?)
VETT2           DB      2*K      DUP(?)
VETT3           DB      3*K      DUP(?)
```

Assemblatori

- **MASM** (Microsoft)
  (*Windows*)
- **TASM** (Borland)
  (*Windows*)
- **NASM**
  (*Windows, Linux, ...*)
- . . . . . .

Modalità di indirizzamento

L'**operando** di un'istruzione può essere:

- in un **registro**
- nell'**istruzione** stessa (operando immediato)
- in **memoria**

Mediante Registro

E la modalità di indirizzamento più **compatto e veloce.**

L'operando è già nella CPU e quindi non è necessario accedere alla memoria

Il registro può essere a 8 o a 16 bit

```
MOV  AX, CX        ;  AX <- CX (16 bit)
```

Il contenuto di CX e' trasferito in AX

Immediato

L'operando è contenuto nell'istruzione

L'accesso all'operando è **veloce** perché l'operando è nella *instruction queue*.

Il dato può essere una costante di 8 o 16 bit

```
MOV  CX, 100 ;  CX <- 100 (16 bit)
```

Trasferisce il valore 100 nel registro CX.
Il valore 100 è memorizzato all'interno dell'istruzione in 2 byte

Indirizzamento Diretto

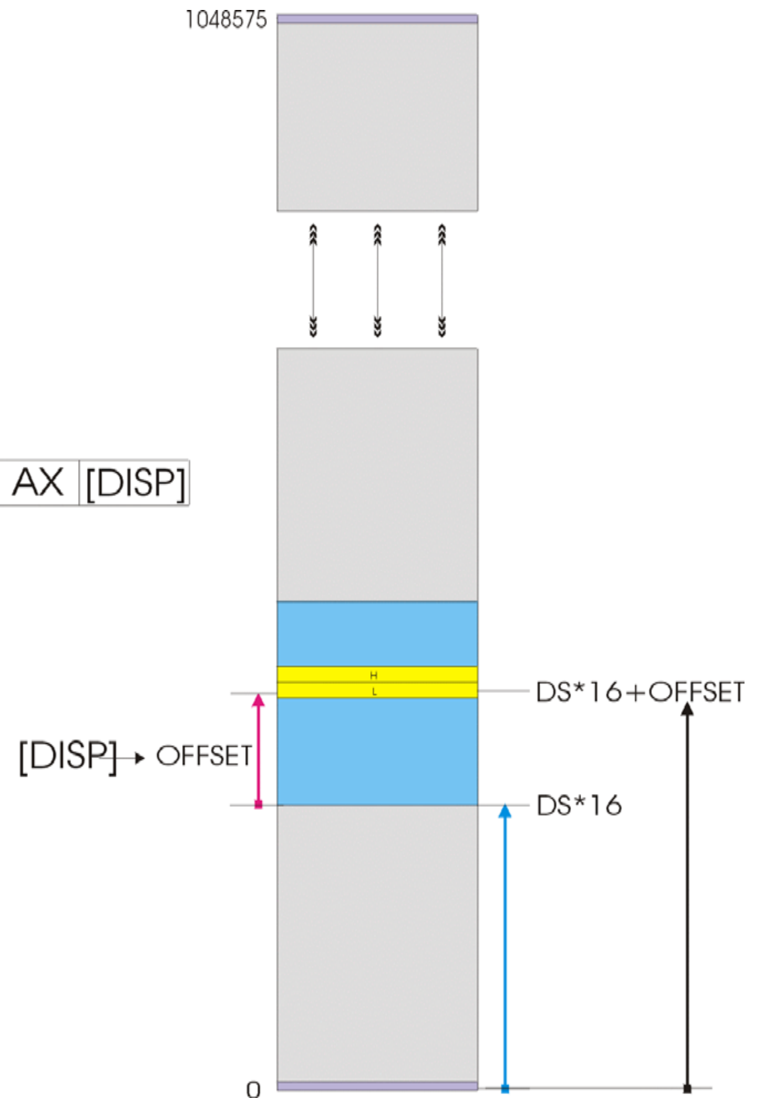
L'EA (Effective Address) è contenuto nel campo displacement dell'istruzione ed è un valore senza segno di 8 o 16 bit. L'indirizzo è prelevato durante la fase di fetch dell'istruzione.

Esempio

```
alfa     DW       0100

MOV  AX,[0100]    ;  AX <- DS:[0100]

MOV  AX, alfa     ;  AX <- DS:[alfa]
```



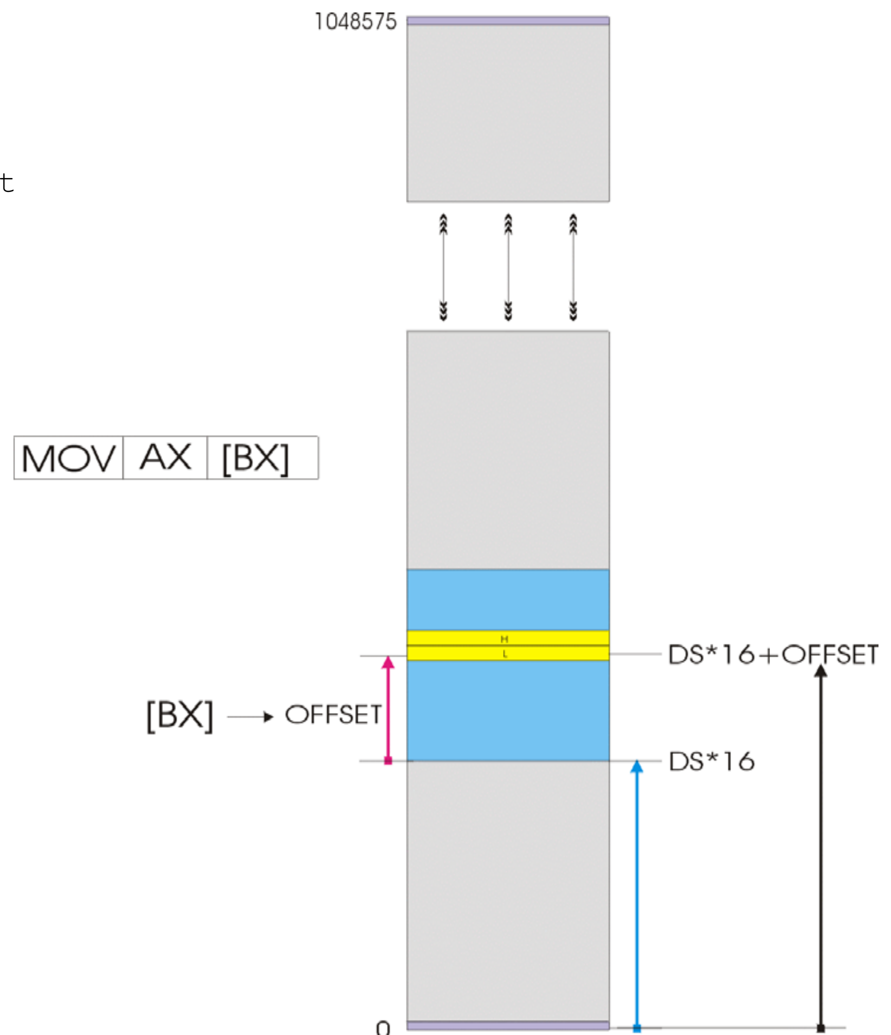
Indirizzamento Indiretto mediante Registro

L'**EA dell'operando è contenuto in un registro base o indice** (BX, BP, DI, SI)

Se si aggiorna in modo opportuno il contenuto del registro, la stessa istruzione può agire su più locazioni di memoria

Con le istruzioni JMP e CALL è possibile utilizzare un qualsiasi registro generale a 16 bit

```
MOV  BX, OFFSET Var ;  BX <- offset di Var
MOV  AX, [BX]       ;  AX <- DS:[BX] (16 bit
```



Indirizzamento Indiretto mediante Registro Base

L'EA si ottiene come somma

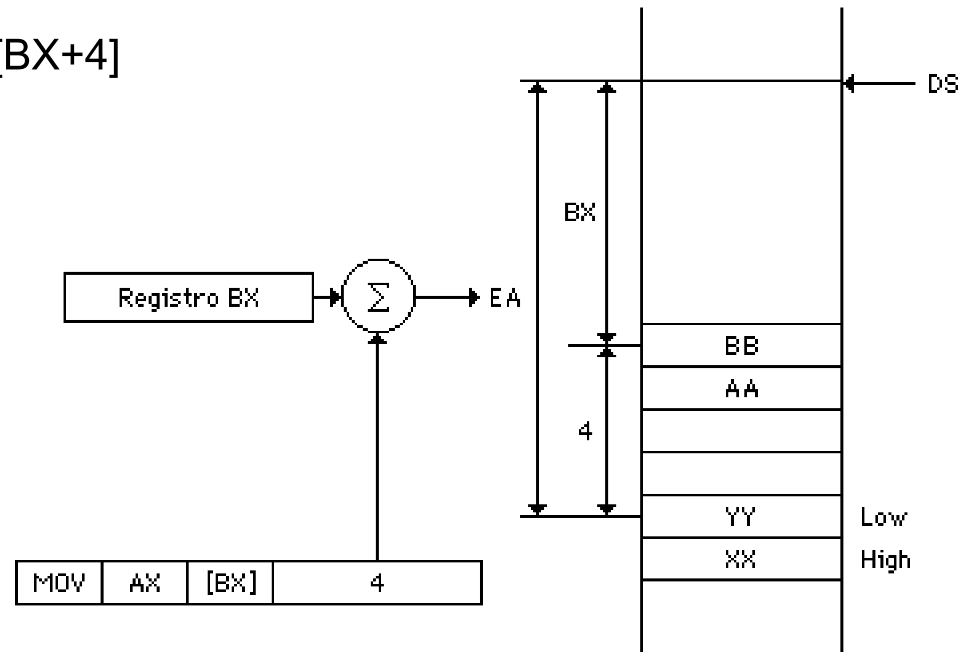
- di un **valore di displacement**
- del **contenuto del registri BX** (segmento DS) o **BP** (segmento SS)

Attenzione: utilizzando il registro BP, il segmento di default non è più DS, bensì SS.

Questa modalità di indirizzamento è utile ad esempio per **accedere allo stesso campo all'interno di una struttura dati composta da più record.**

L'indirizzo di partenza della struttura viene memorizzato nel registro base. Per operare sullo stesso campo di altri record è sufficiente modificare il contenuto del registro base.

MOV AX, [BX+4] ; AX <- DS: [BX+4]

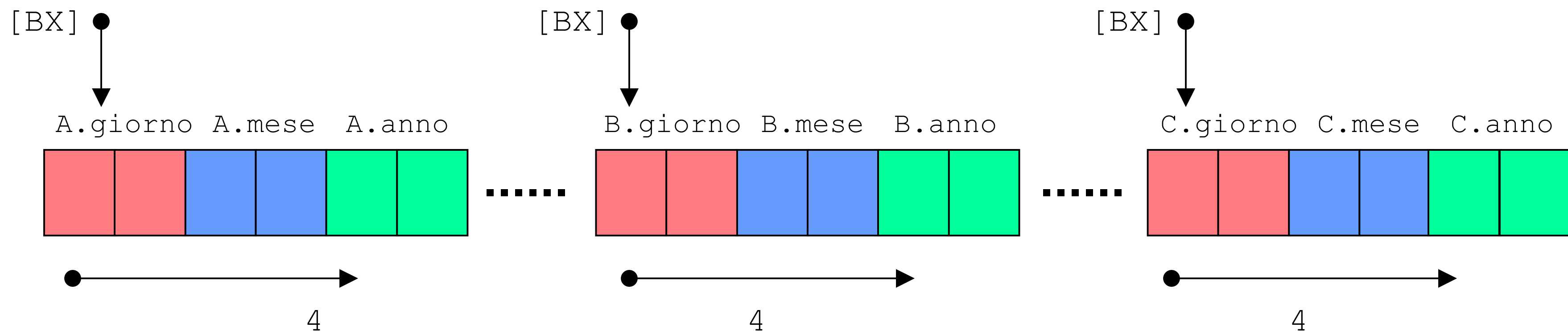


Esempio:

```
typedef struct {
        int giorno,mese,anno;
        } dataType;
dataType A,B,C;
```

A, B e C hanno indirizzi differenti ma ***A.anno***, ***B.anno***, ***C.anno*** hanno lo stesso *displacement* (4) rispetto al primo elemento del record cui appartengono.

Il registro BX contiene un indirizzo nel segmento dati corrente (d1, d2 o d3) per accedere al campo '*anno*' è necessario sommare 4 a tale indirizzo



Indirizzamento Indiretto mediante Registro Indice

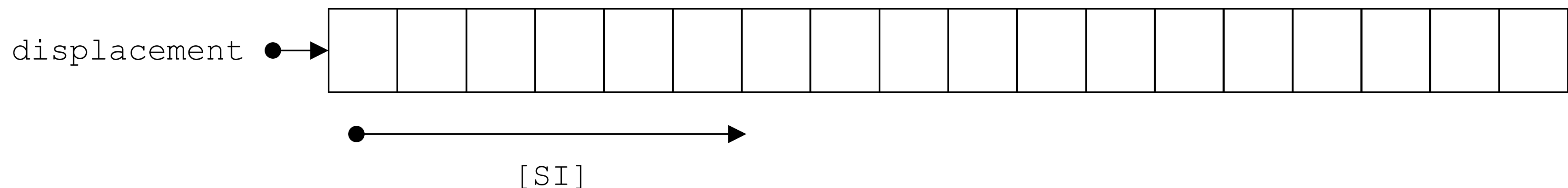
L'EA si ottiene come somma

- di **un valore di displacement**
- del **contenuto di un registro indice** (**SI** o **DI**)

Modalita’ di indirizzamento basata sui registri indice SI o DI ma con caratteristiche analoghe alla modalita’ di indirizzamento mediante registro base vista nel lucido precedente.

Questa modalità di indirizzamento può essere utilizzata ad esempio per **accedere ai diversi elementi di un array**

- il displacement fornisce l'indirizzo di partenza dell'array
- il registro indice seleziona uno degli elementi (il primo elemento viene selezionato quando il registro indice vale 0)



Indirizzamento Indiretto mediante Registro Base e Registro Indice

L'EA si ottiene come somma

- di **un valore di displacement**
- del **contenuto di un registro base** (**BX** o **BP**)
- del **contenuto di un registro indice** (**SI** o **DI**)

Con questa modalità di indirizzamento **due componenti dell'indirizzo possono variare** al momento dell'esecuzione

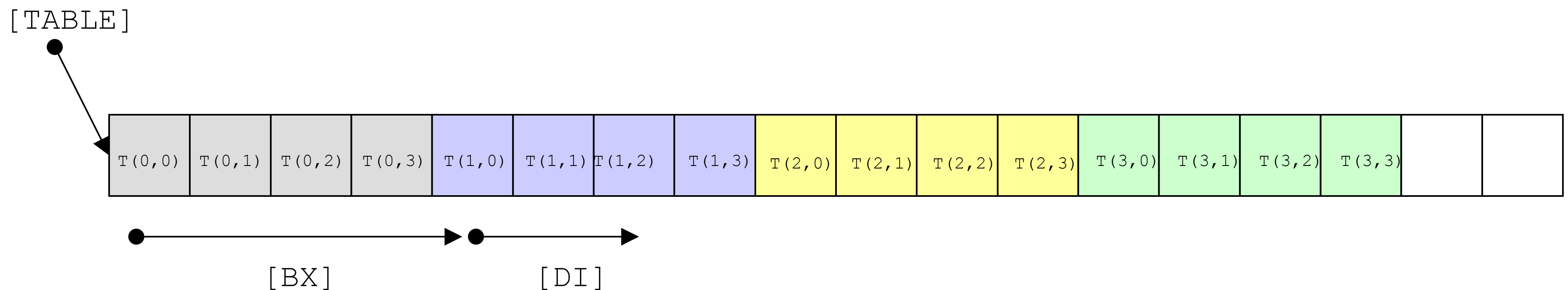
Questa modalità di indirizzamento è utile per accedere, ad esempio, ai diversi elementi di una matrice: il displacement fornisce l'indirizzo di partenza dell'array, mentre il registro base e il registro indice selezionano uno degli elementi (il primo elemento della matrice viene selezionato quando entrambi i registri valgono 0).

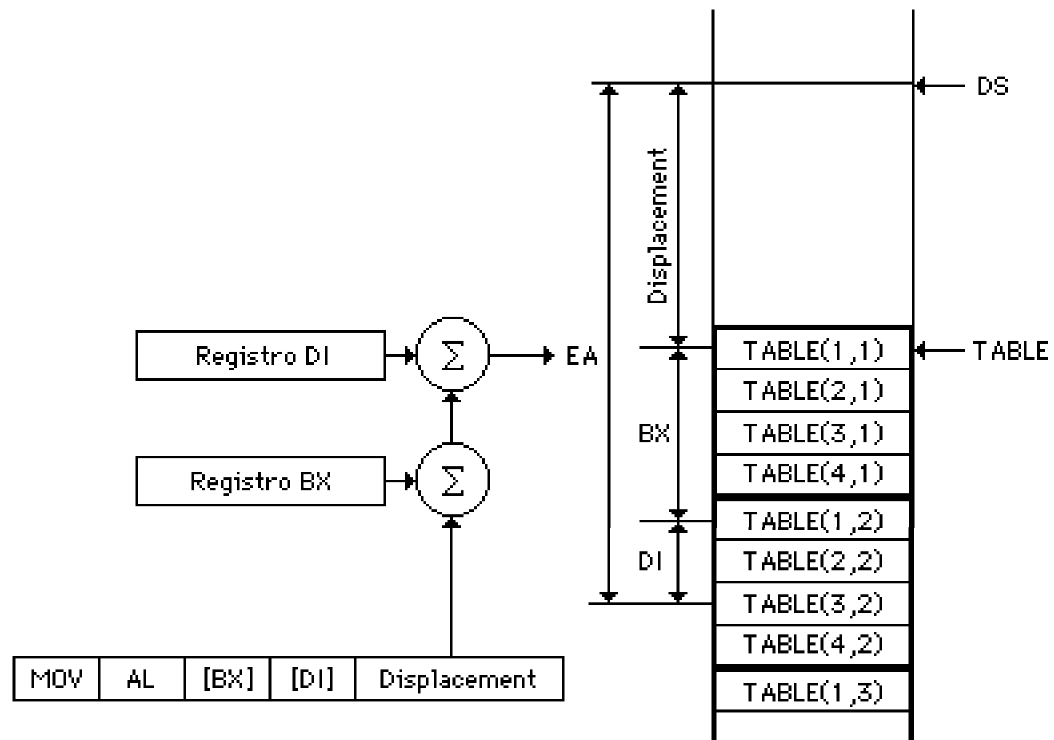
Esempio: TABLE è l'indirizzo simbolico di una matrice di byte 4x4, memorizzata nel data segment corrente

```
MOV BX, 4                ;  inizializza BX a 4
MOV DI, 2                ;  inizializza DI a 2
MOV AL, [TABLE+BX+DI]    ;  AL <- TABLE(1,2)
```

BX contiene il displacement tra l'indirizzo di partenza della matrice e la riga selezionata (la seconda)

DI contiene il displacement tra l'indirizzo di partenza della riga selezionata e la colonna selezionata (la terza)





DS
Displacement
TABLE
Registro DI
EA
BX
Registro BX
DI
TABLE(1,1)
TABLE(2,1)
TABLE(3,1)
TABLE(4,1)
TABLE(1,2)
TABLE(2,2)
TABLE(3,2)
TABLE(4,2)
TABLE(1,3)
MOV
AL
[BX]
[DI]
Displacement

Set di Istruzioni IA16

- **Istruzioni per il trasferimento dati**
- **Istruzioni aritmetiche**
- **Istruzioni per la manipolazione di bit**
- **Istruzioni per il trasferimento del controllo**
- **Istruzioni che operano sulle stringhe**

Istruzioni per il trasferimento dati : MOV (MOVe)

Istruzione di trasferimento dati *general-purpose*

MOV <dest>,<src>

Trasferisce un byte o una word

- **tra due registri**
- **tra un registro e una locazione di memoria**

L'operando sorgente può essere una costante.

Per trasferire un dato da una locazione di memoria a un'altra è necessario utilizzare un registro. Ad esempio per copiare fromVar in toVar,

```
MOV  AX, fromVar ;  copia fromVar in AX
MOV  toVar, AX   ;  e quindi in toVar
```

Non è possibile caricare un valore immediato (come l'indirizzo di un segmento) **in un registro segmento.** Ad esempio per modificare il segmento dati corrente,

```
MOV  AX, segData ;  indirizzo di segData in AX
MOV  DS, AX      ;  e quindi in DS
```

Il registro CS non è utilizzabile come destinazione di un'istruzione MOV.

IN (INput)

Esegue i trasferimenti dati dallo spazio di I/O verso il processore.

IN registro, ind_porta_IO

dove il **registro** deve essere **AX** (word) oppure **AL** (byte).

Questa istruzione consente di trasferire una word o un byte per volta. Se la porta ha indirizzo ≤ 255 si può usare l'indirizzamento diretto

```
IN   AL, 01AH
IN   AX, 080H
```

Se la porta ha indirizzo > 255 si deve usare l'indirizzamento indiretto tramite DX

```
MOV  DX, 8000H
IN   AL, DX
IN   AX, DX
```

OUT (OUTput)

Esegue i trasferimenti dati dal processore verso lo spazio di I/O.

OUT ind_porta_IO, registro

dove il registro deve essere **AX** (word) oppure **AL** (byte).

Questa istruzione consente di trasferire una word o un byte per volta. Se la porta ha indirizzo ≤ 255 si può usare l'indirizzamento diretto

```
OUT  0FFH, AL
OUT  0FFH, AX
```

Se la porta ha indirizzo > 255 si deve usare l'indirizzamento indiretto tramite DX

```
MOV  DX, 0400H
OUT  DX, AL
OUT  DX, AX
```

Esempio: si scriva il codice per comandare (accendere e spegnere) una periferica utilizzando la porta di I/O di indirizzo 0FFH.

```
PORTA   EQU     0FFH    ; definisce l'indirizzo della porta di I/O
MOV  AL, 01H            ;
OUT  PORTA, AL          ; accende la periferica
. . . . . . . .
. . . . . . . .
MOV  AL, 0H             ;
OUT  PORTA, AL ; spegne la periferica
```

La periferica risulta mappata all'indirizzo di I/O FFH. Scrivendo un "1" a questo indirizzo per mezzo della OUT si attivano sia la decodifica dell'indirizzo che il segnale IOWR# la cui contemporaneità causa l'accesso alla periferica. Per lo spegnimento la procedura è analoga e basta caricare uno zero in AL prima di eseguire la OUT. Il segnale di comando per la periferica sarà modificato solo dagli accessi alla porta (indirizzo FFH).

Stack

- area di memoria gestita in modo **LIFO** (**Last In First Out**)
- realizzato **in memoria centrale**
- definito dai registri **SS** e **SP**

In memoria possono coesistere più stack, ognuno al massimo di 64k byte.

Un solo stack è quello “corrente”:

SS contiene l'indirizzo del segmento stack

SP contiene l'offset del top dello stack

BP registro utilizzato per accedere all’area di memoria dello stack senza utilizzare il meccanismo della pila (LIFO)

Lo stack “cresce” dagli indirizzi “alti” a quelli “bassi”.

Le istruzioni che operano sullo stack trasferiscono due byte per volta (una word)

Operazione di **push:**

SP ➔ SP – 2

scrittura di una word al nuovo top

Operazione di **pop:**

lettura di una word dal top

SP ➔ SP + 2

PUSH e **POP**

Operazione di **push** o di **pop** al top dello *stack* selezionato dalla **coppia di registri SS:SP.**

L'unico operando (sempre una word) può essere **un registro o una locazione di memoria.**

PUSHF (PUSH Flags onto stack)

POPF (POP Flags off stack)

Trasferiscono il contenuto del registro *flag* (16 bit) **nello stack e viceversa.**

PUSHF PUSH del registro *flag*

POPF POP del registro *flag*

Le due istruzioni sono necessarie in quanto il registro *flag* non è accessibile direttamente.

Istruzioni per il trasferimento del controllo

CALL (CALL a procedure)
RET (RETurn from procedure)

CALL trasferisce il controllo dal programma chiamante alla procedura chiamata

- salva l'indirizzo di ritorno sullo *stack*
- passa il controllo alla procedura chiamata

RET trasferisce il controllo dalla procedura chiamata al programma chiamantel

- legge dallo stack l'indirizzo di ritorno salvato dalla corrispondente CALL
- ripassa il controllo al chiamante.

Esistono chiamate **near** (all'interno dello stesso segmento di codice, salva solo l'offset di ritorno) e **far** (tra segmenti diversi, deve salvare CS e offset di ritorno - più lenta della precedente).

INT (INT type)

L'istruzione INT consente di invocare una procedura (tipicamente "di sistema") mediante un meccanismo alternativo a quello delle CALL. Tale meccanismo è quello delle interruzioni: il processore serve il tipo di interruzione specificato nell'istruzione

PUSH FLAG
PUSH CS

PUSH IP

*IP <- M[interrupt_type*4]*

*CS <- M[interrupt_type*4+2]*

IRET (Interrupt RETurn)

L'istruzione IRET effettua il ritorno da una procedura attivata mediante il meccanismo delle interruzioni (sia nel caso di interruzione hardware che in quello di interruzione software):

POP IP
POP CS

POP FLAG

JMP (JuMP unconditionally)

Trasferisce il controllo all'istruzione specificata dall'operando, in **modo incondizionato.** Simile all'istruzione di CALL, ma non prevede il rientro nel programma chiamante (*non salva nulla sullo stack*).

L'operando di JMP è simile all'operando dell'istruzione CALL (può essere NEAR o FAR):

```
        JMP  NearLabel
        . . .
NearLabel:
        . . .
```

NearLabel appartiene al segmento codice corrente

```
        JMP  far ptr FarLabel
        . . .
FarLabel  LABEL  far:
        . . .
```

FarLabel appartiene a un segmento codice qualsiasi.

FAR PTR è l'operatore dell'ASM che impone la chiamata o il salto di tipo FAR.

Salti condizionati
Jxxx (Jump conditionally)

Jxxx label

Le istruzioni per il trasferimento condizionato controllano se una condizione è verificata o meno effettuando un test sul registro dei flag:

- **se la condizione è verificata**
  il controllo passa all'istruzione di label
- **se la condizione non è verificata**
  l'esecuzione prosegue con la successiva istruzione

Uso del salto condizionato:

Il salto condizionato è utilizzato per implementare gli operatori relazionali (**==, !=, >, <, >=, <=**).

Ad esempio, per controllare se AX e BX contengono lo stesso valore, si utilizza l'istruzione CMP seguita da un opportuno salto condizionato:

```
CMP  AX, BX;
JE  LabelZero;
```

Per alcuni tipi di test è necessario scegliere tra **due differenti istruzioni di salto**, a seconda che si stia testando il risultato di un'operazione tra **valori con o senza segno.**

"G", "L": greater, less (valori con segno)
"A", "B": above, below (senza segno)

Istruzioni di salto da utilizzare subito dopo l'istruzione CMP, in funzione

- del tipo di test che si desidera effettuare
- del tipo di dato confrontato

Per saltare se	Valori senza segno	Valori con segno
Destinazione $=$ Sorgente	JE	JE
Destinazione $\neq$ Sorgente	JNE	JNE
Destinazione $>$ Sorgente	JA	JG
Destinazione $\geq$ Sorgente	JAE	JGE
Destinazione $<$ Sorgente	JB	JL
Destinazione $\leq$ Sorgente	JBE	JLE

Esempio: realizzazione di un test a tre vie

```
if (A==B) then istruzione_1;  A=B
          else
          if (A<B) then istruzione_2; A<B
          else istruzione_3; A>B
```

```
        CMP  AX, BX
        JAE  LabelGreaterOrEqual ; salta se AX>=BX
        . . .            ; istruzioni eseguite se AX < BX
        . . .
LabelGreaterOrEqual:
        JA LabelGreater ; salta se AX>BX
        . . .            ; istruzioni eseguite se AX = BX

        . . .
LabelGreater:
        . . .            ; istruzioni eseguite se AX > BX

        . . .
```

Esegue **tre differenti blocchi di istruzioni,** in funzione dei valori **senza segno** contenuti nei registri AX e BX. Nel caso di valori **con segno, sostituire JAE con JGE e JA con JG.**

LOOP

LOOP label

Istruzioni per la gestione di cicli che utilizzano implicitamente il registro CX decrementandolo di una unità ad ogni ciclo.

Il ciclo viene ripetuto fino a che il registro CX non è zero.

Esempio:

```
              MOV CX,100     ; numero di iterazioni 100
ciclo:        <istruzione 1>
              <istruzione 2>
              . . . . . . .
              <istruzione n>
              LOOP ciclo     ; il blocco di istruzioni
                             ; è eseguito 100 volte
```

Principali istruzioni Aritmetiche

Le istruzioni aritmetiche prevedono operandi al massimo di 16 bit, in quanto la ALU dell’8086 ha un parallelismo di 16 bit. E’ possibile anche utilizzare operandi di 8 bit.

ADD (ADD) e **ADC** (ADd with Carry)

Sommano 2 operandi di **8** o **16** bit. Uno dei due operandi può risiedere in memoria, l’altro è necessariamente un registro (o una costante, nel caso dell’operando sorgente).

ADD somma l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione:

destinazione = destinazione + sorgente

Convenzione assembler Intel:

```
OP <dst>,<src>         ; <dst> <- <dst> OP <src>
```

ADC effettua la stessa operazione, comprendendo nella somma il flag del riporto (CF) (il cui valore può essere stata modificata da una istruzione aritmetica precedente):

destinazione = destinazione + sorgente + CF

L'istruzione ADC permette di effettuare la somma di numeri maggiori di 16 bit:

Esempio: Somma di due numeri di 32 bit collocati in coppie di registri AX,BX e CX,DX:

```
(AX:BX)   <-   (AX:BX) + (CX:DX)

       ADD  BX, DX    ; somma i 16 bit meno significativi
       ADC  AX, CX    ; somma i 16 bit più significativi
```

La ADD modifica il contenuto del registro dei flag, e in particolare CF che assume il significato di bit di riporto.

SUB (SUBtract) e **SBB** (SuBtract with Borrow)

SUB sottrae l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione:

destinazione = destinazione - sorgente

SBB effettua la stessa operazione, comprendendo nella sottrazione il flag del riporto (CF):

destinazione = destinazione - sorgente - CF

L'istruzione SBB permette di effettuare la sottrazione di numeri maggiori di 16 bit.

Sottrazione di due numeri di 32 bit (AX,BX e CX,DX)

(AX:BX) <- (AX:BX) – (CX:DX)

```
SUB  BX, DX  ; sottrae i 16 bit meno significativi
SBB  AX, CX  ; sottrae i 16 bit più significativi
```

La SUB modifica il contenuto del registro dei flag, e in particolare CF che assume il significato di bit di prestito.

INC (INCrement destination by 1)

Incrementa di 1 il valore del suo operando.
Più veloce e compatta di ADD operando, 1.

DEC (DECrement destination by 1)

Decrementa di 1 il valore dell'operando.

NEG (NEGate)

Cambia segno al suo operando (secondo la rappresentazione in **complemento a 2)**

CMP (CoMPare destination with source)

CMP agisce esattamente come l'istruzione SUB, senza però modificare l'operando destinazione. Lo scopo è quello di modificare il valore di alcuni flag (ved. SUB).

MUL (MULtiply, unsigned) e
IMUL (Integer MULtiply, signed)

MUL <operando>

Se l'operando è **un byte**, viene eseguito:

AX = AL * operando;   (16 bit = 8 bit x 8 bit)

Se l'operando è **una word**, viene eseguito:

DX:AX = AX * operando; (32 bit = 16 bit x 16 bit)

Esistono due distinte istruzioni di moltiplicazione: MUL per i numeri senza segno e IMUL per i numeri con segno (in complemento a 2); l'operazione di moltiplicazione è diversa nei due casi.

Esempio in C:

```
int a, b;

unsigned int c, d;

... a * b ...  →    ... IMUL ...

... c * d ...  →    ... MUL ...
```

Il compilatore C utilizza l'istruzione corretta sulla base del **tipo** degli operandi.

NB: l'operando sorgente non può essere una costante.

DIV (DIVide, unsigned) e
IDIV (Integer DIVide, signed)

DIV <operando>

L'operazione di divisione intera dà luogo a **due** risultati: **quoziente** e **resto** (esistono operatori specifici in C e Pascal). In ASM, una sola istruzione fornisce entrambi i risultati.

Se l'operando è **un byte**, viene eseguito:

```
AL = Quoziente(AX / operando);
AH = Resto(AX / operando);
```

Dimensione degli operandi: 8 bit = 16 bit/8 bit

Se l'operando è **una word**, viene eseguito:

```
AX = Quoziente(AX / operando);
DX = Resto(AX / operando);
```

Dimensione degli operandi: 16 bit = 32 bit/16 bit

DIV: operandi senza segno

IDIV: operandi con segno (compl. a 2)

NB: l'operando sorgente non può essere una costante

Istruzioni per la manipolazione di bit

AND (logical AND)
OR (logical-OR)
XOR (eXclusive-OR)

Eseguono le corrispondenti operazioni logiche tra le coppie di bit di pari posizione degli operandi (corrispondono agli operatori **a bit** del C - &, |, .. - non a quelli **logici** - &&, ||, ..)

```
AND    <dst>,<src>   ;  dst AND src
OR     <dst>,<src>   ;  dst OR src
XOR    <dst>,<src>   ;  dst XOR src
```

TEST

Come l'istruzione AND ma non modifica la destinazione

NOT (logical NOT)

Complementa tutti i bit del suo operando (complemento a 1)

SAL (Shift Arithmetic Left)

- shift verso sinistra di un numero con segno
- segno del numero in CF
- 0 nel bit meno significativo

SAR (Shift Arithmetic Right)

- shift verso destra di un numero con segno
- conserva il segno del numero (bit più significativo)
- pone il bit meno significativo in CF

SHL (SHift logical Left)

- shift verso sinistra di un numero senza segno
- bit più significativo in CF
- 0 nel bit meno significativo

SHR (SHift logical Right)

- shift verso destra di un numero senza segno
- 0 nel bit più significativo
- pone il bit meno significativo in CF